Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 240 (2009) 129–148

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Checking Z Data Refinements Using Traces Refinement

André Didier[1,2], Adalberto Farias[3], Alexandre Mota[4]

Federal University of Pernambuco, Centre of Informatics,
P.O.Box 7851, Cidade Universitria – 50732-970 – Recife – PE – Brazil

Abstract

Data refinement is useful in software development because it allows one to build more *concrete* specifications from *abstract* ones, as long as there is a mathematical relation between them. It has associated rules (proof obligations) that must be discharged; this is normally performed by interactive theorem proving systems. This work proposes an approach based on refinement checking to automatically check the Z data refinement rules. Our approach captures the relational semantics of these rules by using the functional support of CSP_M (the machine-readable version of process algebra CSP) and uses the traceability feature of CSP to find the rules that cannot be satisfied. Moreover, it is able to automatically calculate the mathematical relation between an abstract and a concrete specification, if one exists. We present our approach using an example.

Keywords: Z,data refinement,model-checking

1 Introduction

Although tool support for formal methods has significantly increased in the last years, checking *data refinements* still deserves attention because user intervention is required in many ways. Data refinement techniques aim at making *abstract* specifications more *concrete* by changing their data structures. This requires a relationship between the specifications so that the concrete system *simulates* the abstract one, and generates *refinement rules* (or proof obligations) that must be checked. The user intervention is commonly required to relate the systems and to check such rules. Moreover, depending on the system and on the way the specifications are related, this check becomes tedious or some rule(s) will never be satisfied.

[1] We are thankful to CNPq, which supported us with Grant No. 485488/2006-0. We also thank Dr. Juliano Iyoda for suggestions in early drafts of this paper and Professor Jim Woodcock for substantial help on the final version of this paper.

[2] alrd@cin.ufpe.br

[3] acf@cin.ufpe.br

[4] acm@cin.ufpe.br

doi:10.1016/j.entcs.2009.05.049

Recently, model checkers and SAT solvers have been employed to check data refinements [1,12,13]. However, this use still presents some limitations concerned with input/output manipulation, infinite state-spaces handling or with the construction of the mathematical relation between the abstract and the concrete system. In another direction, theorem provers handle input/output and infinite state-spaces but requires user intervention, in general [8].

This work proposes a way of automatically checking data refinements for Z [16] (introduced in Section 2) and also of calculating the relation between the analysed specifications (abstract and concrete), as long as it exists. We briefly introduce our target notation CSP_M (the machine-readable version of the process algebra CSP [11]) in Section 3 and show how to use its functional support to describe the Z refinement rules. Then, we propose a template process to discharge them; the process performs a specific sequence of events if and only if all rules are satisfied (detailed in Section 4). We analyse this template process using the refinement checker FDR [7]; as this check is automatic, theorem proving is unnecessary.

The translation from Z to CSP_M follows the strategy implemented in a support tool [5]. Some adjustments are necessary as we deal with refinement rules instead of the behaviour of Z specifications. Moreover, the translation has limitations because some Z constructs might not have a direct correspondent in CSP_M. Nevertheless, all solutions in this direction present similar limitations [2,3,15].

We want to stress that we do *not* use the equivalence between data refinement and process refinement, as reported in [2]. Instead, we describe the proof obligations according to their relational semantics and check them. If any of them fails, we use the traceability feature provided by CSP_M and FDR to find the rules that are not satisfied, in such a way that the proposed refinement (concrete system) can be adjusted. This occurred in our example. Moreover, as we use refinement checking, our strategy gives a result only when both specifications have finite state spaces. The use of compression techniques like data abstraction [9] is a further effort to overcome this, as pointed out as future work in Section 6.

The general contribution of this paper concerns the reduction of user intervention when employing data refinement in the development process. Currently, user intervention is necessary only to provide the relation (as a Z schema) between the abstract and the concrete specifications, and to translate all schemas into CSP_M. Our specific contributions are: (*i*) the usage of CSP to capture the relational semantics of the Z data refinement rules; (*ii*) the automatic verification of Z data refinements given a retrieve relation, and; (*iii*) the automatic calculation of a retrieve relation, if one exists, to justify data refinement.

In Section 5 we discuss related work and in Section 6 we present our conclusions and potential directions for future work.

2 Overview of Z

The language Z is based on set theory and first-order logic. It provides abstract data types like sets, sequences and bags, and the *schema* language, which is in-

tended to structure and compose descriptions. A Z schema is useful to describe state, initialisation and operations (viewed as relations), which contain an enabling condition (the *precondition*) and an associated effect (*post-condition*).

In the *blocking* view of Z, preconditions act as *guards* for operations. Thus, as long as the precondition of an operation holds in the *before* state, the *after* state is calculated as described in the post-condition; otherwise, an undefined state is originated. On the other hand, the *non-blocking* view establishes that, when a precondition holds, its corresponding operation yields an after state according to the post-condition; otherwise, it yields an arbitrary state [16] (anything can happen). In this work we adopt the *blocking* view and present Z using the vending machine example given in [16] (Fig. 1). We abstract away the payment, and the kind of drink that gets dispensed.

$$Status ::= yes \mid no$$
$$Digit == 0..9$$
$$\mathrm{seq}_3[X] == \{s : \mathrm{seq}\, X \mid \#s = 3\}$$

$$State_A \;\hat{=}\; [busy : Status;\ vend : Status]$$

$$Init_A \;\hat{=}\; [State'_A \mid busy' = no \land vend' = no]$$

$$Vend_A \;\hat{=}\; [\Delta State_A;\ o! : Status \mid busy' = no \land o! = vend]$$

$$Choose \;\hat{=}\; [\Delta State_A;\ i? : \mathrm{seq}_3\, Digit \mid busy = no \land busy' = yes]$$

Fig. 1. Abstract specification of a vending machine

The machine uses three types. The free-type *Status* is used to signal the success (*yes*) or failure (*no*) of the current interaction, and to keep track of the progress of a transaction. The type *Digit* is a number between 0 and 9; and $\mathrm{seq}_3[X]$ is a sequence of three elements of type X. The state ($State_A$) contains two Boolean variables to indicate if the machine is in use (*busy*), and if the current transaction is successful (*vend*). The initialisation ($Init_A$) sets both state variables to false ($busy'$ means the next value of *busy*).

Users interact with the machine through the operation *Choose*, which inputs a three-digit sequence ($i? : \mathrm{seq}_3\, Digit$) at once and then dispenses the drink, as long as the numbers were correctly chosen. The operation $Vend_A$ simply signals whether the transaction is successful or not (through the nondeterministic output $o!$) and makes the machine available again ($busy' = no$). Note that, in the operations *Choose* and $Vend_A$, the variable $vend'$ is left undetermined because its value is nondeterministically chosen.

In the concrete vending machine proposed by [16], the state ($State_C$) contains the amount of digits; that is, $State_C \mathrel{\widehat{=}} [digits : 0..3]$ (the digits are entered sequentially).

This allows one to relate the two states through a *Retrieve* relation in such a way that, if the system is not in use, then no digits have been entered.

$$\begin{array}{|l} \mathit{Retrieve} \\ \hline \mathit{State}_A \\ \mathit{State}_C \\ \hline busy = no \Leftrightarrow digits = 0 \\ \hline \end{array}$$

The remaining schemas are illustrated in Fig. 2. The initialisation ($Init_C$) considers that no digit has been entered. Thus, the machine becomes *busy* after one execution of *FirstPunch* and in the two following executions of *NextPunch*. After that, it executes $Vend_C$, dispensing the drink and outputting (nondeterministically) its status.

$$\begin{array}{|l} \mathit{Init}_C \\ \hline \mathit{State}'_C \\ \hline digits' = 0 \\ \hline \end{array}$$

$$\begin{array}{|l} \mathit{FirstPunch} \\ \hline \Delta \mathit{State}_C \\ d? : \mathit{Digit} \\ \hline digits = 0 \land digits' = 1 \\ \hline \end{array}$$

$$\begin{array}{|l} \mathit{NextPunch} \\ \hline \Delta \mathit{State}_C \\ d? : \mathit{Digit} \\ \hline (0 < digits < 3 \land digits' = digits + 1) \\ \lor (digits = 0 \land digits' = digits) \\ \hline \end{array}$$

$$\begin{array}{|l} \mathit{Vend}_C \\ \hline \Delta \mathit{State}_C \\ o! : \mathit{Status} \\ \hline digits' = 0 \\ \hline \end{array}$$

Fig. 2. Concrete specification of the vending machine

According to [16], a concrete operation *COp* refines an abstract one *AOp* ($AOp \sqsubseteq COp$) if and only if *COp simulates AOp*, in the sense that everything *COp* does, *AOp* also does, possibly more nondeterministically. This correspondence requires a relation between the concrete and the abstract states in two possible directions: *forward* (from abstract to concrete) or *backward* (from concrete to abstract). The conceptual difference between them concerns the resolution of nondeterminism. In this paper we use the latter because the concrete specification postpones the nondeterminism to the end of the execution of $Vend_C$ and the refinement cannot be proved using *forward* simulation. Moreover, in most data refinements, each abstract operation has one direct concrete version. Our example is quite different: *FirstPunch* refines *Choose* and $Vend_C$ refines $Vend_A$. As *NextPunch* preserves the abstract state, it refines $\Xi State_A$, (ΞS stands for the preservation of S) [16].

The proof obligations originated by the data refinement use some operators over sets and relations. Table 1 shows the main operators.

The rules formalising the notion of a backward simulation between Z specifications involve initialisation, applicability and correctness of operations (Definition 2.1). We use the relational version of the refinement rules proposed in [14]; they extend those proposed in [16] to also contemplate inputs and outputs. We just

Table 1
Operators

Operator	Explanation
$\mathrm{id}(T)$	The identity over a type; it is given by $\mathrm{id}(T) = \{(x, x) \mid x \in T\}$
$\mathrm{dom}\, R$	The domain of a relation; it is given by $\mathrm{dom}\, R = \{x \mid (x, y) \in R\}$
$\mathrm{ran}\, R$	The range of a relation; it is given by $\mathrm{ran}\, R = \{y \mid (x, y) \in R\}$
$s \lhd R$	Domain restriction; it is given by $s \lhd R = \{(x, y) \mid (x, y) \in R \land x \in s\}$
$s \lhd\!\!\!- R$	Domain subtraction; it is given by $s \lhd\!\!\!- R = \{(x, y) \mid (x, y) \in R \land x \notin s\}$
$R \rhd s$	Range subtraction; it is given by $R \rhd s = \{(x, y) \mid (x, y) \in R \land y \notin s\}$
$R_1 \parallel R_2$	The parallel (separate and simultaneous) application of R_1 and R_2
$R_1 \fatsemi R_2$	Composition; it is given by $R_1 \fatsemi R_2 = \{(x, z) \mid (x, y) \in R_1 \land (y, z) \in R_2\}$. If R_1 is unary, $R_1 \fatsemi R_2 = \{y \mid x \in R_1 \land (x, y) \in R_2\}$
$\overline{S}$	The complement of a set in its type. For example, if S is of type $\mathbb{P}\, X$, then $\overline{S} = \{x : X \mid x \notin S\}$

replace the correctness rule with the suitable version for the blocking view, as presented in [1]. Moreover, since our system does not explicitly initialises inputs, and as we assume that all observations of the state come from system's outputs, we need a smaller set of rules. We represent a Z specification as a triple $(State, Init, \{Op_i\}_{i \in I})$ containing a state schema $(State : Exp)$, an initialisation schema $(Init)$ and an indexed set of operation schemas $(\{Op_i\}_{i \in I})$. Each operation is a relation of type $State \times State_{in} \leftrightarrow State \times State_{out}$, where $State_{in}$ $(State_{out})$ represents the inputs (outputs) parameters of the operation. This requires the existence of retrieve relations for the state (R), inputs (R_{in}) and outputs (R_{out}).

Definition 2.1 Let $(CS, CI, \{COp_i\}_{i \in I})$ and $(AS, AI, \{AOp_i\}_{i \in I})$ be two Z specifications. $(CS, CI, \{COp_i\}_{i \in I})$ is a backward simulation of $(AS, AI, \{AOp_i\}_{i \in I})$ with respect to the retrieve relations R, R_{in} and R_{out} if and only if:

- $CI \fatsemi R \subseteq AI$ (**b-init**)
- $\mathrm{dom}\ COp_i \subseteq \mathrm{dom}((R \parallel R_{in}) \rhd (\mathrm{dom}\ AOp_i))$ (**b-app**)
- $COp_i \fatsemi (R \parallel R_{out}) \subseteq (R \parallel R_{in}) \fatsemi AOp_i$ (**b-corr**)

Each rule in the above definition has an interpretation in the relational semantics. The initialisation (***b-init***) must be checked once and establishes that for each concrete initial state there is a corresponding abstract initial state. The applicability (***b-app***) and the correctness (***b-corr***) rules must be checked for all operations. The former says that whenever it is possible to perform the abstract operation AOp_i, it must be possible to perform the concrete operation COp_i on the corresponding concrete state and concrete input. The latter establishes that whenever it is possible to perform the abstract operation, and the corresponding concrete operation

can result in state C' and output $C_!$, then it must be possible to find an abstract state A' and output $A_!$, corresponding to that C' and $C_!$, which is the result of performing the abstract operation.

We point out that the original rules of data refinement consider total relations (operations and retrieves). However, as Z specifications usually present partial operations, the totalisation is achieved by using augmented domains in such a way that: (*i*) the state includes a new bottom element (that is, $State_\perp = State \cup \{\perp\}$, where $\perp \notin State$); (*ii*) the operations are totalised to map elements outside its precondition (that is, $\dot{op} = op \cup \{x : State_\perp \mid x \notin \mathrm{dom}\, Op \bullet (x, \perp)\}$); and (*iii*) the retrieve is *lifted* to also propagate undefinedness (that is, $\mathring{R} = R \cup \{(\perp_{State}, \perp_{State^A})\}$). Fig. 3 illustrates the totalisation of an operation and of a retrieve relation.

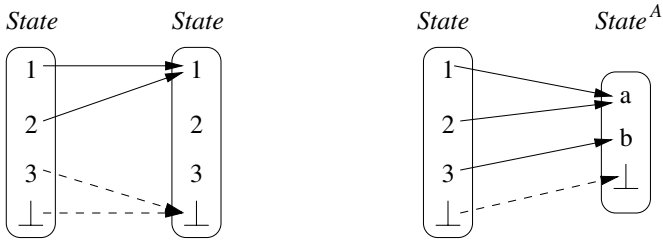


Fig. 3. Totalisation of $\{1 \mapsto 1, 2 \mapsto 1\}$ and lifting of $\{1 \mapsto a, 2 \mapsto a, 3 \mapsto b\}$

Note that the rules of Definition 2.1 do not involve totalised relations. Actually, the use of domain restriction and subtraction allows one to find equivalent rules for partial relations [16]. Furthermore, the relations R_{in} and R_{out} depend on the compared operations. Thus, regarding *Choose* and *FirstPunch*, R_{out} disappears in ***b-corr*** (outputs are absent), whereas R_{in} is given by:

$$
\begin{array}{|l}
R_{in} \\
s : \mathrm{seq}_3\, Digit \\
d : Digit \\
\hline
d = head(s) \\
\hline
\end{array}
$$

On the other hand, the operations *VendA* and *VendC* have the same outputs but no inputs. Hence, R_{in} disappears in ***b-app*** and in ***b-corr***, and $R_{out} = \mathrm{id}(Status)$.

Concerning the operations *NextPunch* and $\Xi State_A$, we need some adjustments to apply ***b-app*** and ***b-corr***. Note that *NextPunch* has an input $d? : Digit$, whereas $\Xi State_A$ has not. In order to represent all operations uniformly, we define a new operation $XiState_A$ that is similar to $\Xi State_A$ but with an abstract input of type $\mathrm{seq}_3\, Digit$, which is not manipulated by $XiState_A$. As *NextPunch* is intended to read the second and the third digits, we use a new retrieve relation for inputs:

$$XiState_A \quad \Xi State_A \quad i? : \mathrm{seq}_3\ Digit$$

$$R_{in} \quad i : \mathrm{seq}_3\ Digit \quad d : Digit \quad d = head(tail(i)) \lor d = head(tail(tail(i)))$$

In [16], the refinement of the vending machine used the identity to map existing inputs and outputs parameters. This simplification is possible because the domains are finite. Moreover, no input parameter was inserted into $\Xi State$ because the refinement rules were adapted to ignore it. We inserted an input in $XiState_A$ to avoid changes in the refinement rules. In specification meanings, the new input is only used to represent operations uniformly and does not affect the state preservation.

3 Overview of CSP

The process algebra CSP [11] is suitable for describing the behaviour of systems. It has constructs for modelling successful termination, deadlocks, livelocks and atomic computations (modelled as *events*). Its machine-readable version CSP_M also provides functional features and support for manipulating integers, sequences, sets, booleans and customized types. We concentrate on showing only the elements of CSP_M used in this work; a complete description can be found in [7,11]. Furthermore, CSP_M code is represented using `true type` fonts.

Processes interact with each other by communicating events through channels (declared with the keyword `channel`). A channel with a type defines a family of events, whereas a non-typed channel defines only one non-communicating event.

Concerning the Boolean expressions, we use the operators `not` (negation), `and` (conjunction) and `or` (disjunction). We also use sequences, sets and some basic operators: `empty(A)` tests if the set `A` is empty, `member(e,A)` tests if `e` is an element of `A`, `diff(A,B)` gives the set difference between `A` and `B`, `s1^s2` concatenates the sequence `s1` with `s2`, `head(<a>^s)` gives the head element `a`, and `tail(<a>^s)` returns the tail sequence `s`.

Sets and sequences over a type `T` are defined as `Set T` and `Seq T`, respectively. Alternatively, sets can be defined using ranges or comprehension. For example, `{1..5}` or `{x|x <- {1..10}, x <= 5}` represent the same set `{1,2,3,4,5}`. The comma in the predicate part of the comprehension is a shorthand for logical conjunction, and `x <- {1..10}` means that the value `x` is taken from the set `{1..10}`.

The conditional construction `if b then E1 else E2` is also available in CSP_M, where `b` is a Boolean condition and `E1` and `E2` are expressions of any (the same) type. The operators `==` and `<=` are overloaded and may be used for integers, sequences, sets and tuples. For sets, the operator `<=` means inclusion ($\subseteq$).

Pattern matching is also supported. For example, the first element of a pair is easily obtained by using the function `first`. The underscore '_' matches any value.

```
first((x,_)) = x
```

Custom data types can be defined using the keyword `datatype`. For example, the type *Status* can be defined as:

```
datatype Status = yes | no
```

The abstract state space ($State_A$) is determined by the set containing all pairs involving the type `Status`; that is, {`(busy,vend) | busy <- Status, vend <- Status`}. This corresponds to the set {`(no,no),(no,yes),(yes,no),(yes,yes)`}.

Another functional feature is local definition (the `let ... within` construct). For instance, the function `getFirst` below returns the first component of an abstract state (represented by a pair).

```
getFirst(state) = let
        (busy,vend) = state
    within busy
```

3.1 Behaviour in CSP_M

In CSP_M, processes can be primitive (or basic) or defined using operators. The following grammar defines the subset of CSP_M we are interested in.

Event ::= ChanName | ChanName ? Variable | ChanName ! Expression

Process ::= `STOP` | `SKIP` (basic processes)
| Event -> Process (prefix)
| Process [] Process (external choice)
| Process ; Process (sequential composition)

An Event can be a channel name (ChanName), an input event (ChanName ? Variable) or an output event (ChanName ! Expression). Variable is an identifier (a name of a variable).

The process `STOP` denotes deadlock (abnormal termination) whereas `SKIP` means successful termination. The process `e -> P` is ready to engage on the event `e`; after performing `e`, it behaves like `P`. The process `P [] Q` behaves like `P` or `Q`, depending on the other processes (the environment) it interacts with. The process `P;Q` represents the sequential composition of `P` and `Q`; it requires that `P` terminates successfully.

In this work, we study processes using the semantic model of traces (or $\mathcal{T}$, for short), where a process is represented by the set of all sequences of events (traces) it can perform [11]. For example, the process `P = a -> b -> STOP` is represented by {`<>,<a>,<a,b>`}, where `<>` means `P` performed no event yet, `<a>` means `P` performed the event `a`, and `<a,b>` means `P` performed `a` followed by `b`. The set of all traces of a process P is denoted by $\mathcal{T}(P)$. We have chosen this model because it is sufficient for our purposes. In $\mathcal{T}$, the notion of refinement is given in terms of set inclusion: a process Q refines a process P if and only if the traces of Q are a subset of those of P; that is, $P \sqsubseteq_{\mathcal{T}} Q \Leftrightarrow \mathcal{T}(Q) \subseteq \mathcal{T}(P)$. Thus, the process `Q = a -> STOP` refines `P` because {`<>,<a>`} $\subseteq$ {`<>,<a>,<a,b>`}. The refinement $P \sqsubseteq_{\mathcal{T}} Q$ is put into the refinement checker FDR by the statement `assert P [T= Q`.

Besides performing refinement checks, FDR provides traceability when a refinement fails (the counterexample). We use this feature to find the rule (and operation) that failed when checking a Z data refinement, as explained in the next section.

4 Automatic Verification of Z Data Refinement

Our approach to automatically check a Z data refinement is illustrated in Fig. 4. The parameters `AZSpec` and `CZSpec` represent the abstract and the concrete specifications, respectively. Both of them, conjointly with the retrieve relations `R`, `R_in` and `R_out` are translated into CSP_M before performing the check. The proof obligations are encoded as functions. Due to the pattern matching feature provided by CSP_M, we are also able to automatically compute the retrieve relation to validate a data refinement between two given specifications, as long as it exists.

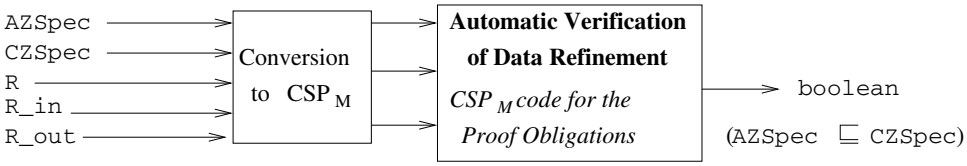


Fig. 4. Overall strategy

4.1 Translating Z into CSP_M

The state is represented by a tuple; the state space, the initialisation and the operations are converted into set comprehensions. Although some operations might not have input/output (as the operations *Choose* and $Vend_A$ in Fig. 1), we represent them uniformly in the conventional type $State \times State_{in} \leftrightarrow State \times State_{out}$. To achieve that, we insert the necessary input/output (of type `UNDEFINED`) in the schemas (see the sets `Choose` and `VendA` in Table 2).

```
datatype UNDEFINED = {BOTTOM}
```

Table 2 shows the complete translation of the abstract vending machine. We point out that, CSP_M does not provide a direct type corresponding to a sequence of three digits ($\text{seq}_3\ Digit$) in Z. Thus, we define the type `SeqDigit_3` as follows:

```
SeqDigit_3 = {<x,y,z>|x <- Digit,y <- Digit,z <- Digit}
```

Table 3 shows the translation of the concrete vending machine. Recall from Section 2 that *Retrieve* considers only the component *busy* to relate the states. As CSP_M does not have operators for $\Leftrightarrow$ and $\Rightarrow$, we use the equivalences $A \Leftrightarrow B \equiv A \Rightarrow B \wedge B \Rightarrow A$ and $A \Rightarrow B \equiv \neg A \vee B$ from predicate calculus, and pattern matching to implement them as the functions `iff` and `implies`, respectively. Then, we use the function `iff` to define an implementation for the `Retrieve` relation.

```
implies(A,B) = not(A) or B
iff(A,B) = implies(A,B) and implies(B,A)
Retrieve = {((busy,vend),digits) | (busy,vend) <- StateA,
                        digits <- StateC, iff(busy == no,digits == 0)}
```

The association of R_{in} and R_{out} with each pair of operation is achieved by using the abstract operations as a suffix. Thus, `R_in_Choose` and `R_out_Choose` are associated with the pair (*Choose*, *FirstPunch*), and so on.

Table 2
Translation of the abstract vending machine

Z Definitions	CSP_M
$Status ::= yes \mid no$	`datatype Status = yes | no`
$Digit == 0..9$	`Digit = {0..9}`
$State_A$ $busy, vend : Status$	`StateA = {(busy,vend)| busy <- Status, vend <- Status}`
$Init_A$ $State'_A$ $busy' = no \land vend' = no$	`InitA = {(no, no)}`
$Choose$ $\Delta State_A$ $i? : \text{seq}_3\ Digit$ $busy = no \land busy' = yes$	`Choose={(((busy,vend),in), ((busy',vend'),out))| (busy,vend)<-StateA,(busy',vend')<-StateA, in <- SeqDigit_3, out <- UNDEFINED, busy == no, busy' == yes}`
$Vend_A$ $\Delta State_A$ $o! : Status$ $busy' = no$ $o! = vend$	`VendA = { (((busy,vend),in), ((busy',vend'),out)) | (busy,vend)<-StateA,(busy',vend')<-StateA, in<-UNDEFINED,out<-Status, busy' == no, out == vend }`

```
R_in_Choose = {(d,i) | d <- Digit, i <- SeqDigit_3, head(i) == d}
R_out_Choose = id(UNDEFINED)

R_in_XiStateA = {(d,i) | d <- Digit, i <- SeqDigit_3,
                         d = head(tail(i)) or d = head(tail(tail(i)))}
R_out_XiStateA = id(Status)

R_in_VendA = id(UNDEFINED)
R_out_VendA = id(Status)
```

The translation of the proof obligations is almost a direct transcription to CSP_M according to the prefixed functions of Table 4. The translation of all rules of Definition 2.1 is illustrated in Table 5.

4.2 Discharging Proof Obligations Automatically

In CSP_M, a specification is represented as a triple `(State,Init,Ops)`, where `Ops` is a sequence of operations. Thus, both versions of the vending machine are repre-

Table 3
Translation of the concrete vending machine

Z Definitions	CSP_M
$State_C$ $digits : 0..3$	`StateC = { digits | digits <- {0..3} }`
$Init_C$ $State'_C$ $digits' = 0$	`InitC = { 0 }`

$FirstPunch$
$\Delta State_C$
$d? : Digit$
$digits = 0 \land digits' = 1$
	`FirstPunch = { ((digits,in),(digits',out)) |`
	`  digits <- StateC, digits' <- StateC,`
	`  in <- Digit, out <- UNDEFINED,`
	`  digits == 0 and digits' == 1 }`

$NextPunch$
$\Delta State_C$
$d? : Digit$
$(0 < digits < 3 \land$
$digits' = digits + 1) \lor$
$((digits = 0) \land$
$digits' = digits)$
	`NextPunch = { ((digits,in), (digits',out)) |`
	`  digits <- StateC, digits' <- StateC,`
	`  in <- Digit, out <- UNDEFINED,`
	`  (0 < digits and digits < 3 and`
	`  digits' == digits + 1) or`
	`  ((digits == 0) and`
	`  digits' == digits) }`

$Vend_C$
$\Delta State_C$
$o! : Status$
$digits' = 0$
	`VendC = { ((digits,in), (digits',out)) |`
	`  digits <- StateC, digits' <- StateC,`
	`  in <- UNDEFINED, out <- status,`
	`  digits' == 0 }`

sented by:

```
AZSpec = (StateA,InitA,<Choose,VendA,XiStateA>)
CZSpec = (StateC,InitC,<FirstPunch,VendC,NextPunch>)
```

The pairs of operations to be compared are directly obtained from `AZSpec` and `CZSpec`, following the same order they appear in the sequences. Moreover, as we capture invalid rules using traces, we need to define special events.

```
channel initOk, initNotOk
channel appOk,appNotOk:AOPNAME.COPNAME
channel corrOk,corrNotOk:AOPNAME.COPNAME
```

Channels `initOk` and `initNotOk` define non-communicating events that are performed when ***b-init*** is valid or not, respectively. Channels `appOk` and `appNotOk`, and

Table 4
Translation of the main Z operators

Operator	**Translation to CSP_M**
$\mathrm{id}(T)$	`id(T)={(x,x) | x <- T}`
$\mathrm{ran}\, R$	`ran(R)={y | (x,y) <- R}`
$\mathrm{dom}\, R$	`dom(R)={x | (x,y) <- R}`
$s \lhd R$	`dres(A,R)={(x,y) | (x,y) <- R,member(x,A)}`
$s \lhd\!\!\!\cdot\, R$	`ndres(A,R)={(x,y) | (x,y) <- R, not member(x,A)}`
$R \rhd s$	`nrres(R,B)={(x,y) | (x,y)<-R, not member(y,B)}`
$R_1 \parallel R_2$	`prll(R,S)={((w,x),(y,z)) | (w,y) <- R, (x,z) <- S}`
$\overline{s}$	`diff(X,s)` is the set difference `X` $\setminus$ `s` where `s` is of type $\mathbb{P}$`X`.
$R \fatsemi S$	`comp(R,S) = {(x,z) | (x,y) <- R, (y,z) <- S}`. If R is unary, the composition $R \fatsemi S$ is also a unary relation given by `comp_un(R,S) = {y | x <- R, (x,y) <- S}`

Table 5
Translation of the backwards simulation rules

Rule	**Z**	CSP_M
b-init	$CI \fatsemi R \subseteq AI$	`comp_un(CI,R) <= AI`
b-app	$\mathrm{dom}\, COp \subseteq$ $\mathrm{dom}((R \Vert R_{in}) \rhd (\mathrm{dom}\, AOp))$	`diff({(cs,in) | cs <- CS, in<-inputs(COp)}, dom(COp)) <= dom(nrres(prll(R,R_in),dom(AOp)))`
b-corr	$COp_i \fatsemi (R \parallel R_{out}) \subseteq$ $(R \parallel R_{in}) \fatsemi AOp_i$	`comp(COp, prll(R,R_out)) <= comp(prll(R,R_in), AOp)`

`corrOk` and `corrNotOk` have the same purpose for ***b-app*** and ***b-corr***, respectively. As these rules are applied to a pair of operations, the corresponding channels accept to communicate the names of the compared operations: abstract and concrete, in this order. Thus, instead of directly communicating the CSP_M representation of the operations on channels, we communicate only their names, which are defined in new data types: `AOPNAME` for abstract operations and `COPNAME` for concrete ones. The sequences `AOpNames` and `COpNames` represent the name of the operations that must be communicated on channels. Note that the names follow the same order the corresponding operations appear in the specifications.

```
datatype AOPNAME = ChooseOp | XiStateAOp | VendAOp
datatype COPNAME = FirstPunchOp | NextPunchOp | VendCOp
```

```
AOpNames = <ChooseOp,VendAOp,XiStateAOp>
COpNames = <FirstPunchOp,VendCOp,NextPunchOp>
```

To use the suitable retrieve in a refinement rule, we put them into sequences that follow the same order as the operations.

```
R_in_Sequence = <R_in_Choose,R_in_VendA,R_in_XiStateA>
R_out_Sequence = <R_out_Choose,R_out_VendA,R_out_XiStateA>
```

In the process representation of a refinement rule, each function of Table 5 is used to enable the associated events. The process `initialisation`, for example, receives both initialisations as parameters and uses the global `Retrieve` between them. It performs `initOk` as long as `b-init` is valid (`b-init` i the CSP_M representation of **b-init**); otherwise, it performs `initNotOk`.

```
initialisation(Inits) = let
     (AI,CI) = Inits
     R = Retrieve
   within if b_init(AI,CI,R) then
             initOk -> SKIP
          else initNotOk -> STOP
```

In its base case, the process `applicability` behaves like `SKIP`. In the inductive case, it receives the concrete state and five sequences: abstract operations, concrete operations, names of abstract operations, names of concrete operations and retrieves for inputs. Operations and names are taken from their respective sequences (heads) and the global `Retrieve` is used to to relate the states. The process checks the rule `b-app` (the CSP_M representation of **b-app**) for the first pair of operations. If `b-app` returns true, the process communicates the names of the operations on channel `appOk` and recursively checks the remaining operations; otherwise, the names are communicated on channel `appNotOk`, ending with a `STOP`.

```
applicability(_,<>,_,_,_,_) = SKIP
applicability(CS,AOpSeq,COpSeq,AOpSeqName,COpSeqName,R_in_Seq) = let
        AOp = head(AOpSeq)
        COp = head(COpSeq)
        AOpName = head(AOpSeqName)
        COpName = head(COpSeqName)
        R = Retrieve
        R_in = head(R_in_Seq)
   within if b_app(CS,AOp,COp,R,R_in) then
            appOk.AOpName.COpName ->
             applicability(CS,tail(AOpSeq),tail(COpSeq),
              tail(AOpSeqName),tail(COpSeqName),tail(R_in_Seq))
          else
           appNotOk.AOpName.COpName -> STOP
```

The process `correctness` is defined similarly to `applicability`. The difference is related to some parameters, the use of the function `b-corr` (the CSP_M

representation of **b-corr**), and the use of channels `corrOk` and `corrNotOk`.

```
correctness(<>,_,_,_,_,_) = SKIP
correctness(AOpSeq,COpSeq,AOpSeqName,COpSeqName,R_in_Seq,R_out_Seq)=
    let
      AOp = head(AOpSeq)
      COp = head(COpSeq)
      AOpName = head(AOpSeqName)
      COpName = head(COpSeqName)
      R = Retrieve
      R_in = head(R_in_Seq)
      R_out= head(R_out_Seq)

    within if b_corr(AOp,COp,R,R_in,R_out) then
            corrOk.AOpName.COpName ->
             correctness(tail(AOpSeq),tail(COpSeq),tail(AOpSeqName),
               tail(COpSeqName),tail(R_in_Seq),tail(R_out_Seq))
           else
            corrNotOk.AOpName.COpName -> STOP
```

The complete set of backward rules is represented by the process `BackwardRules`, which is defined as the sequential composition of the previous processes.

```
BackwardRules(CS,AI,CI,AOpSeq,COpSeq,
              AOpSeqName,COpSeqName,R_in_Seq,R_out_Seq) =
 initialisation(AI,CI);
  applicability(CS,AOpSeq,COpSeq,AOpSeqName,COpSeqName,R_in_Seq);
   correctness(AOpSeq,COpSeq,AOpSeqName,COpSeqName,R_in_Seq,R_out_Seq)
```

As long as the processes `initialisation`, `applicability` and `correctness` successfully terminate, `BackwardRules` performs the sequence

$$\texttt{<initOk>\^{}<appOk.AOpName}_i\texttt{.COpName}_i\texttt{>}^i\texttt{\^{}<corrOk.AOpName}_i\texttt{.COpName}_i\texttt{>}^i$$

where $\texttt{<ch.aname}_k\texttt{.cname}_k\texttt{>}^k$ means $\texttt{<ch.aname}_1\texttt{.cname}_1\texttt{,...,ch.aname}_k\texttt{.cname}_k\texttt{>}$ and `k = #AOpSeq` (the sequences of operations have the same size and order). For initialisation purposes, we define the process `BACK_RULES` that extracts the necessary parameters from the global structures and passes them to `BackwardRules`, as long as the retrieves are total. The totality is checked by testing if the domain of each retrieve relation is equal to the respective concrete domain. If this is invalid for at least one retrieve, the process `BACK_RULES` performs the event `partialRetrieve` and then deadlocks. Thus, when using some partial retrieve to check a refinement in our approach, FDR produces the trace `<partialRetrieve>`.

```
BACK_RULES = let
    (_,AI,AOpSeq) = AZSpec
    (CS,CI,COpSeq) = CZSpec
    R_in_Seq = R_in_Sequence
    R_out_Seq = R_out_Sequence
```

```
  within if ((StateC != dom(Retrieve)) or
             (Digit != dom(R_in_Choose)) or
             (UNDEFINED != dom(R_out_Choose)) or
             (Digit != dom(R_in_XiStateA)) or
             (Status != dom(R_out_XiStateA)) or
             (UNDEFINED != dom(R_in_VendA)) or
             (Status != dom(R_out_VendA)))
           then partialRetrieve -> STOP
         else BackwardRules(CS,AI,CI,AOpSeq,COpSeq,AOpNames,
                        COpNames,R_in_Seq,R_out_Seq)
```

To validate a data refinement using refinement checking we define a process that performs the sequence of events in case of success: `RefOk`. It uses auxiliary processes that communicate events denoting success for each rule. `RefInitOk` performs `initOk` behaves like `SKIP`. When the sequence of operations is empty, `RefAppOk` and `RefCorrOk` behave like `SKIP`; otherwise, they communicate the names of the first operations (head) on their respective channels and consider the remaining operations (tail).

```
RefOk = RefInitOk;RefAppOk(AOpNames,COpNames);
          RefCorrOk(AOpNames,COpNames)

RefInitOk = initOk -> SKIP

RefAppOk(<>,_) = SKIP
RefAppOk(AOpSeqName,COpSeqName) =
        appOk.head(AOpSeqName).head(COpSeqName) ->
           RefAppOk(tail(AOpSeqName),tail(COpSeqName))

RefCorrOk(<>,_) = SKIP
RefCorrOk(AOpSeqName,COpSeqName) =
        corrOk.head(AOpSeqName).head(COpSeqName) ->
           RefCorrOk(tail(AOpSeqName),tail(COpSeqName))
```

Finally, the validation is put into FDR by using the refinement assertion:

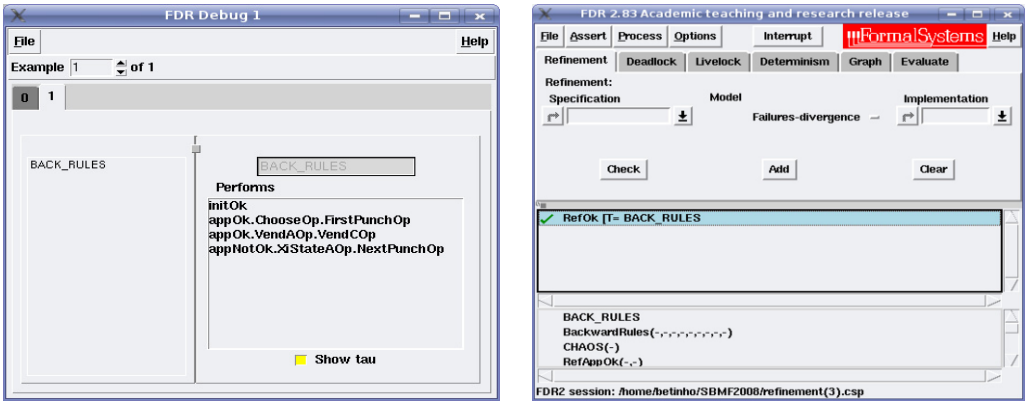
```
assert RefOk [T= BACK_RULES
```

If `RefOk [T= BACK_RULES` is valid, then `BACK_RULES` performs the trace representing the success of applying the rules of Definition 2.1.

Note that the traces refinement here does not take into account the existence or inexistence of nondetermism nor undefinedness. They are dealt into the proof obligation rules which were fully translated into CSP_M. Then, the `BACK_RULES` process actually simulates the verification of the proof obligation rules. Thus, our approach does not verify a data refinement between Z specifications using their corresponding process representations [1,2,4]. Instead is performs the real validation of Z data refinement into a model checker, like a human being does to discharge the proof obligations.

4.3 Trace Inspection in Invalid Refinements

The refinement check `assert RefOk [T= BACK_RULES` is our CSP_M encoding of the refinement reported in [16]. We performed it using FDR and surprisingly obtained a false result. Then, we used the sequence of performed events (the counterexample) illustrated in Fig. 5.a to see which rule and operation was invalid. The occurrence of `appNotOk.XiStateAOp.NextPunchOp` reveals that *NextPunch* is not a valid simulation for $XiState_A$. Then, we performed a deeper investigation and found a subtlety: *NextPunch* is not defined when $digits = 3$. Although this never occurs, the specification must consider such a situation. Then, we adjusted the precondition of *NextPunch* from $digits = 0$ to $digits = 0 \vee digits = 3$ and obtained a valid refinement (Fig. 5.b).



(a) (b)

Fig. 5. Traceability of FDR

4.4 Automatic Computation of Retrieves

In the previous section, the functions implementing the proof obligations receive the retrieve relations as parameters. In this section we show how to determine a retrieve automatically, if it exists. We consider only the state but inputs and outputs are dealt with similarly. The existence of such relations means the refinement is valid (Equation 1).

$$\exists\, R \in \mathbb{P}\, R_{max} \Rightarrow A \sqsubseteq_R C \tag{1}$$

In the above equation, A and C are the abstract and the concrete specification, respectively, $\sqsubseteq_R$ means 'refined with respect to R', and R_{max} is the cartesian product between the abstract and the concrete states. The set comprehension `RMax` below implements the relation R_{max} above.

```
RMax = {(sc,sa) | sc<-StateC, sa <-StateA}
```

The function `valid_retrieves` takes the abstract and the concrete specifications and returns the set of all retrieve relations that satisfy all rules in the conjunction of the application of `b_init`, `b_app_all` and `b_corr_all`, where `b_app_all` and

b_corr_all are new functions that apply b_app and b_corr, respectively, to all operations.

```
valid_retrieves(ASpec,CSpec) = let
    (_,AI,AOpSeq) = ASpec
    (CS,CI,COpSeq) = CSpec
 within { R | R <- Set(RMax),b_init(AI,CI,R) and
  b_app_all(CS,AOpSeq,COpSeq,AOpNames,COpNames,R,R_in_Sequence) and
  b_corr_all(AOpSeq,COpSeq,AOpSeqName,
                                COpSeqName,R,R_in_Seq,R_out_Sequence)}

b_app_all(_,<>,_,_,_,_,_) = true
b_app_all(CS,AOpSeq,COpSeq,AOpSeqName,COpSeqName,R,R_in_Seq) = let
        AOp = head(AOpSeq)
        COp = head(COpSeq)
        AOpName = head(AOpSeqName)
        COpName = head(COpSeqName)
        R_in = head(R_in_Seq)
   within b_app(CS,AOp,COp,R,R_in) and
          b_app_all(CS,tail(AOpSeq),tail(COpSeq),tail(AOpSeqName),
                    tail(COpSeqName),R,tail(R_in_Seq))

b_corr_all(<>,_,_,_,_,_,_) = true
b_corr_all(AOpSeq,COpSeq,AOpSeqName,COpSeqName,R,R_in_Seq,R_out_Seq)=
 let
     AOp = head(AOpSeq)
     COp = head(COpSeq)
     AOpName = head(AOpSeqName)
     COpName = head(COpSeqName)
     R_in = head(R_in_Seq)
     R_out= head(R_out_Seq)
 within b_corr(AOp,COp,R,R_in,R_out) and
        b_corr_all(tail(AOpSeq),tail(COpSeq),tail(AOpSeqName),
                   tail(COpSeqName),R,tail(R_in_Seq),tail(R_out_Seq))
```

The relation R is calculated by extension, following the principle behind model checking (exhaustive search). Thus, all possible sets of combinations between concrete and abstract states are covered, without user intervention and with the guarantee that a retrieve is found as long as there is a refinement. In order to validate the refinement, we only need to check if the set valid_retrieves(.) is not empty. To do that, we use the process EXIST_RETRV and the refinement assertion as follows:

```
EXIST_RETRV(ASpec,CSpec) =
   not empty(valid_retrieves(ASpec,CSpec)) & SKIP

assert EXIST_RETRV(AZSpec,CZSpec) [T= SKIP
```

Theorem 4.1 associates the above check with the existence of a retrieve relation.

Theorem 4.1 *Let* `AZSpec` *and* `CZSpec` *an abstract and a concrete Z specifications, respectively, and let* `ACSPSpec` *and* `CCSPSpec` *be their CSP_M representations. Then,* `EXIST_RETRIEVE(ACSPSpec,CCSPSpec) [T= SKIP` *if and only if* `CZSpec` *refines* `AZSpec` *with respect to a retrieve relation* `R`*.*

Proof

From the CSP theory [11], `EXIST_RETRV(ACSPSpec,CCSPSpec) [T= SKIP` is valid only if `valid_retrieves(ACSPSpec,CCSPSpec)` returns a non-empty set. This happens only if the functions `b_init`, `b_app_all` and `b_corr_all` return true. From this and from Equation 1 we know that there exists a retrieve relation that validates the refinement. Conversely, if the refinement is valid, then there is a retrieve `R` that validates `b_init`, `b_app_all` and `b_corr_all`. Therefore, `valid_retrieves(ACSPSpec,CCSPSpec)` returns a non-empty set and then `EXIST_RETRV(ACSPSpec,CCSPSpec) [T= SKIP` is valid.

The complete CSP_M code of our example can be requested via e-mail or downloaded from http://www.cin.ufpe.br/~alrd/entcs2008/refinements.csp.

5 Related Work

The approach presented in [2] shows the correspondence between data and process refinements using the CSP semantics of Z. Thus, a Z data refinement is valid, if and only if, the refinement between the corresponding CSP processes is valid. We see our work as complementary, as we follow the relational semantics of Z data refinements. We use processes only to establish a behaviour for each proof obligation instead of the behaviour of a Z specification. This is more useful to find out, via counterexamples, which rule invalidated a refinement. In principle, both approaches deal with the same class of problems and have the same limitation when domains are infinite.

In [1] the conversion from Z to Alloy enables the use of SAT solvers to verify Z data refinements. Like our approach the user does not need to provide a retrieve relation a priori; it can be computed automatically. As Alloy is very close to Z, the conversion used in [1] is, in principle, simpler than ours. Moreover, the way Alloy deals with finite domains is more efficient than FDR. Nevertheless, Alloy does not provide traceability features to capture rules and operations that invalidated a refinement. A common limitation of both approaches concerns infinite state space systems. The use of data abstraction [6,9] in both approaches would be helpful to limit the scope of data domains.

A similar approach for verifying Z data refinements using model checking is proposed in [4], where Z specifications are translated into equivalent processes so that process refinement corresponds to data refinement. That work is able to handle infinite domains in communications by using special schemas to make them finite. These schemas cannot contain any relation between outputs and state variables. Our approach is free of such a restriction as it allows inputs/outputs refinement through retrieve relations.

As far as we know, the use of process refinement does not provide detailed information about an invalid rule [1,2,4]. Instead, in an invalid refinement, one can observe, for example, a sequence of operations performed only by the concrete system that is not performed by the abstract one. In our approach, invalid refinements reveal exactly what rule and operation failed. This allows adjustments in the original models, as usually employed in counterexample-guided approaches.

The technique reported in [12] uses classical model checking (and temporal logic) to verify data refinements based on forward simulations. Another translation strategy is proposed to write the transitions and paths as structures that can be analysed by a model checker. The strategy requires that the retrieve relation is given and suggests the embedding of input/output into the state. Our approach covers two situations: when a retrieve relation is given and when it is not. In the first situation, we indicate if the refinement is valid or not. In the second situation we are able to find (if it exists) a retrieve relation to validate a refinement.

6 Conclusions

Combining data refinement with stepwise development is a powerful alternative to establish correctness between specifications, where transformations are applied to derive more concrete artifacts that are mathematically equivalent to the original ones. Because such a guarantee is usually based on theorem proving, user intervention might be required to verify the underlying proof obligations. In this context, the use of techniques and tools to make data refinement as automatic as possible is essential for its practical application.

In this paper we proposed an approach for automatically checking Z data refinements. We consider the relational semantics of Z [16] and the functional support of CSP_M to write the proof obligations as functions that basically check set inclusions. Then, we provided template processes whose refinement check is valid if and only if the proof obligations are satisfied. Although FDR is not the best tool to check our functions, it provides traceability features that allows one to find invalid rules and discards the use of theorem proving. Three immediate results emerge from this: (*i*) automatic verification of a data refinement between two specifications; (*ii*) automatic calculation of a retrieve relation that assures a data refinement; and (*iii*) the use of counterexamples for adjusting the specification whenever the refinement between them is invalid. We used (*iii*) to detect a subtlety in a common example of the literature.

As refinement checking is limited to finite state space systems, our technique

cannot deal with the state space explosion directly. Nevertheless, by integrating our approach with data abstraction [6,9] we can limit the data domains (to finite but sufficient subsets of them) before applying our approach. This is a topic for future work. Furthermore, we performed the translation to CSP_M by hand. However, we intend to add this feature in the tool presented in [10]. Currently, the tool is able to translate Z specifications into processes. This new feature will discharge the user of manipulating the CSP_M code of the proof obligations.

References

[1] Bolton, C., *Using the alloy analyzer to verify data refinement in Z.*, Electr. Notes Theor. Comput. Sci. **137** (2005), pp. 23–44.

[2] Bolton, C., J. Davies and J. Woodcock, *On the refinement and simulation of data types and processes*, in: *IFM99*, 1999, pp. 273–292.

[3] Borba, P. and S. Meira, *From VDM specifications to functional prototypes*, Journal of Systems and Software **21(3)** (1991), pp. 267–278.

[4] Derrick, J. and H. Wehrheim, *On Using Data Abstractions for Model Checking Refinements*, Acta Informatica **44** (2007), pp. 41–71.

[5] Farias, A., A. Mota and A. Sampaio, *From CSP_Z to CSP_M: A transformational java tool*, in: *WMF*, 2001, pp. 1–10.

[6] Farias, A., A. Mota and A. Sampaio, *Efficient CSP_Z data abstraction*, in: E. B., J. Derrick and G. Smith, editors, *Integrated Formal Methods (IFM 2004)*, Lecture Notes in Computer Science **2999** (2004), pp. 108–127.

[7] Goldsmith, M., "FDR: User Manual and Tutorial, version 2.77," Formal Systems (Europe) Ltd (2001).

[8] Larrecq, J. and I. Mackie, "Proof Theory and Automated Deduction," Applied Logic Series **6**, Kluwer Academic Publishers, 1997.

[9] Mota, A., P. Borba and A. Sampaio, *Mechanical abstraction of CSP_Z processes*, in: L. Eriksson and P. Lindsay, editors, *Formal Methods Europe (FME'2002)*, LNCS **2391**, 2002, pp. 163–183.

[10] Mota, A. and A. Sampaio, *Model checking CSP_Z: strategy, tool support and industrial application*, Science of Computer Programming **40** (2001), pp. 59–96.

[11] Roscoe, A., "The Theory and Practice of Concurrency," Prentice Hall, 1998.

[12] Smith, G. and J. Derrick, *Verifying data refinements using a model checker*, Form. Asp. Comput. **18** (2006), pp. 264–287.

[13] Smith, G. and K. Winter, *Proving temporal properties of z specifications using abstraction*, in: *ZB*, 2003, pp. 260–279.

[14] Stepney, S., D. Cooper and J. Woodcock, *More powerful z data refinement: Pushing the state of the art in industrial refinement*, in: *ZUM'98: Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation* (1998), pp. 284–307.

[15] Wehrheim, H., *Data abstraction for CSP-OZ*, , **1709**, LNCS (1999).

[16] Woodcock, J. and J. Davies, "Using Z: specification, refinement, and proof," Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.